

Subject: Coroutines (hopefully done)
From: Van Snyder
Reference: 03-258r1, section 1.1; 04-149r1, 04-345

1 **1 Number**

2 TBD

3 **2 Title**

4 Coroutines.

5 **3 Submitted By**

6 J3

7 **4 Status**

8 For consideration.

9 **5 Basic Functionality**

10 Provide for coroutines.

11 **6 Rationale**

12 In many cases when a “library” procedure needs access to user-provided code, the user-provided code
13 needs access to entities of which the library procedure is unaware. There are at least four ways by which
14 the user-provided code can gain access to these entities:

- 15 • The user-provided code can be implemented as a procedure that is invoked either directly or by
16 way of a dummy procedure, the extra entities can be made public entities of some module, and
17 accessed in the user-provided procedure by use association.
- 18 • The user-provided code can be implemented as a procedure that is invoked either directly or by
19 way of a dummy procedure, and the extra entities can be put into common if they’re data objects.
- 20 • The user-provided code can be implemented as a procedure that takes a dummy argument of
21 extensible type, which procedure is invoked either directly or by way of a dummy procedure, and
22 the extra entities can be put into an extension of that type.
- 23 • The library procedure can provide for *reverse communication*, that is, when it needs access to user-
24 provided code it returns instead of calling a procedure. When the user-provided code reinvokes
25 the library procedure, it somehow finds its way back to the appropriate place.

26 Each of these solutions has drawbacks. Entities that are needlessly public increase maintenance expense.
27 The maintenance expense of common is well known. If the user-provided procedure expects to find its
28 extra information in an extension of the type of an argument passed through the library procedure, the
29 dummy argument has to be polymorphic, and the user-provided code has to execute a SELECT TYPE
30 construct to access the extension. Reverse communication causes a mess that requires GO TO statements
31 to resume the library procedure where it left off, which in turn requires to simulate conventional control
32 structures using GO TO statements. This reduces reliability and increases development and maintenance
33 costs.

34 Reverse communication is, however, a blunt-force simulation of a well-behaved control structure that
35 has been well-known to computer scientists for decades: The *coroutine*. Coroutines would allow user-
36 provided code needed by library procedures more easily to gain access to entities of which the library

1 procedure is unaware, without causing the disruption of the control structure of the library procedure
2 that reverse communication now causes.

3 I polled users of my library software for solutions of ordinary differential equations (both initial value
4 and boundary value), evaluation of integrals by quadrature (in one and several dimensions), nonlinear
5 least squares, nonlinear zero finding, and nonlinear optimization.

6 All of these packages provide for both forward (“call a subroutine”) and reverse (“return when access to
7 user code is needed”) communication. The latter is a coroutine, heretofore implemented without syn-
8 tactic support. The lack of syntactic support makes a mess of the control structure of these procedures.

9 I asked the users these questions:

- 10 1. Do you use forward or reverse communication?
- 11 2. On some arbitrary scale of your own devising, rate your model as “simple” or “complicated.”
- 12 3. If you presently use reverse communication, and I revise my software to require you to use forward
13 communication, and explain excellent new features in Fortran 2003 to support getting extra (non-
14 state) parameters into your model, will it cause trouble for you?
- 15 4. If I were to revise my software so that in your reverse communication loop you would need to
16 replace a CALL statement with a new RESUME statement, would it cause trouble for you?

17 Roughly half the users answered question #1 with “reverse.” Of those, roughly 80% answered question
18 #2 with “complicated.” Almost all of the users who answered questions #1 and #2 in that way answered
19 question #3 with, in essence, “Fu** off!” All of those who answered question #3 in that way answered
20 question #4 with “not a problem.”

21 Intrinsic support for coroutines would allow me to replace the internal control structures of my library
22 routines that provide for reverse communication with ones that are far clearer and easier to understand,
23 thereby reducing my long-term maintenance costs, without causing substantial cost for my users.

24 Coroutines are also useful to implement *iterators*, which are procedures that can be used both to enumer-
25 ate the elements of a data structure and to control iteration of a loop that is processing those elements.
26 Without coroutines, the way this is usually supported is to put the loop body into a subroutine, and
27 pass that subroutine’s name to the iterator. The problem with this is that it increases both development
28 and maintenance costs. The subroutine that implements the loop body can’t be an internal subroutine
29 so one must either bundle up everything the loop references and put it into an extension of the type
30 of some object the iterator passes through to the subroutine, or make everything more global than it
31 deserves to be. Of course, these considerations apply equally to the user code needed by quadrature etc.
32 software, but the case of a loop body makes the undesirability of packaging it as a separate subroutine
33 more obvious.

34 **7 Estimated Impact**

35 Small. Minor additions to Section 12. Estimated at J3 meeting 169 to be at 5 on the JKR scale.

36 **8 Detailed Specification**

37 Provide two new statements, which we shall here call SUSPEND and RESUME.

38 Provide a new form of subroutine, the *coroutine*, that cannot be recursive, cannot contain an ENTRY
39 statement, and is the only subprogram in which a SUSPEND statement is allowed.

40 If a coroutine suspends its execution by executing a SUSPEND statement, and its execution is subse-
41 quently resumed by executing a RESUME statement, execution resumes after the SUSPEND statement.
42 Otherwise (either execution of the subroutine was terminated by execution of a RETURN or END state-
43 ment, or it was invoked by a CALL statement), execution continues with the first executable construct
44 of the invoked subroutine.

1 It is nonsense to allow a SUSPEND statement in a function, because there's no way to RESUME a
 2 function.
 3 The RESUME statement need not appear in the same subprogram as the CALL statement that initiated
 4 execution of the coroutine.
 5 It is not necessary or useful to prohibit internal subroutines to be coroutines. In fact, it's more difficult
 6 to do so.
 7 Coroutines should be allowed to be type-bound procedures, actual arguments and procedure pointer
 8 targets. Generic coroutines should be allowed. It's more difficult to prohibit these than to allow them.
 9 It is proposed at this time that only data entities that have the SAVE attribute retain their definition
 10 status and value upon execution of a SUSPEND statement. Similarly, nonsaved entities in modules, or
 11 nonsaved common blocks, accessed from a coroutine, might become undefined if no other scoping units
 12 are accessing them after a SUSPEND statement is executed. This could be changed later, including in
 13 a future revision of the standard, without invalidating any program.
 14 If the activation record does not continue to exist after execution of a SUSPEND statement, a SUSPEND
 15 statement is prohibited to appear within a DO construct that has *loop-control* consisting of *do-variable*
 16 = *scalar-int-expr*, *scalar-int-expr*[, *scalar-int-expr*]. This could also be changed later.
 17 Argument association information does not survive execution of a SUSPEND statement. New actual
 18 arguments shall be supplied on the RESUME statement.
 19 No matter whether the entire activation record survives a SUSPEND statement or not, the specification
 20 part needs to be elaborated upon resumption, at least to recreate automatic objects, because their extents
 21 and length parameters could depend on common variables or variables accessed by use association (or
 22 host association in the case of internal coroutines).
 23 Fortran already has a limited form of coroutine: The relation between an input/output item list and a
 24 format is a coroutine relation. So it's not an entirely new concept for Fortran.

25 8.1 Suggested edits

26 The following edits are intended to illustrate the magnitude of the project.

27	or <i>resume-stmt</i>	11:28+
28	or <i>suspend-stmt</i>	11:31+
29	[Editor: Replace “When ... point.” by the following paragraph:]	15:6-7
30	If execution of a procedure defined by a coroutine subprogram is completed by execution of a SUSPEND	
31	statement, and it is subsequently invoked by executing a RESUME statement while the same instance	
32	of its host as existed when the SUSPEND statement was executed exists, execution begins with the	
33	first executable construct following the SUSPEND statement that completed execution of the coroutine.	
34	Otherwise, when any procedure is invoked, execution begins with the first executable construct appearing	
35	after the invoked entry point.	
36	[Editor: Insert “, SUSPEND” after “RETURN”.]	15:9
37	[Editor: Replace “execution ... statement” by “completion of execution of an instance of a procedure”.]	15:9
38	[Editor: Replace “execution ... statement” by “completion of execution of an instance of a procedure”.]	16:4+3
39	[Editor: Replace “execution ... statement” by “completion of execution of an instance of a procedure”.]	84:7
40	[Editor: Replace “execution ... statement” by “completion of execution of an instance of a procedure”.]	84:11
41	[Editor: Replace “a ... in” by “completion of execution of an instance”.]	84:15
42	[Editor: Replace “a RETURN ... executed” by “execution of an instance of a procedure is completed”.]	91:26-27
43	[Editor: Replace “Execution ... statement” by “Completion of execution of an instance of a procedure”.]	100:20
44	[Editor: Replace “a procedure ... statement” by “an instance of a procedure is completed”.]	115:5

1	[Editor: Replace “execution ... statement” by “completion of execution of an instance of a procedure”.]	115:9-10
2	[Editor: Add a comma at the end of each item in the list.]	168:12-19
3	(5 $\frac{1}{2}$) Execution of a SUSPEND statement within the range of the do construct; Execution of a	168:19+
4	RESUME statement might reactivate the do construct (12.7.4), or	
5	[Editor: Replace “statement or an” by “, SUSPEND or”.]	249:16-17
6	[Editor: Replace “or subroutine” by “, subroutine, or coroutine”.]	256:17
7	R1218 $\frac{1}{2}$ <i>resume-stmt</i> is RESUME <i>procedure-designator</i> [([<i>actual-arg-spec-list</i>])]	266:19+
8	C1222 $\frac{1}{2}$ (R1218 $\frac{1}{2}$) The <i>procedure-designator</i> shall designate a coroutine subprogram.	
9	[Editor: After “ <i>prefix</i> ” insert “[COROUTINE]”.]	282:8
10	C1251 $\frac{1}{3}$ (R1232) If COROUTINE appears, <i>prefix</i> shall not include RECURSIVE.	282:19+
11	If COROUTINE appears, the subprogram is a coroutine subprogram (12.7.4).	282:29+
12	C1251 $\frac{2}{3}$ (R1235) An ENTRY statement shall not appear in a coroutine subprogram.	283:4+
13	12.5.2.5$\frac{1}{2}$ SUSPEND statement	284:34+
14	R1237 $\frac{1}{2}$ <i>suspend-stmt</i> is SUSPEND	
15	C1259 $\frac{1}{3}$ (R1237 $\frac{1}{2}$) A SUSPEND statement is allowed only in the scoping unit of a coroutine subprogram.	
16	C1259 $\frac{2}{3}$ (R1237 $\frac{1}{2}$) A SUSPEND statement shall not appear within a do construct in which the <i>do-stmt</i>	
17	has <i>loop-control</i> consisting of <i>do-variable</i> = <i>scalar-int-expr</i> , <i>scalar-int-expr</i> [, <i>scalar-int-expr</i>].	
18	Execution of a SUSPEND statement completes execution of the instance of the subprogram in which it	
19	appears. Execution may be resumed at the executable construct following the SUSPEND statement by	
20	executing a RESUME statement (12.7.4).	
21	12.7.4 Coroutines	289:13+
22	A coroutine subprogram is a subroutine in which COROUTINE appears in its SUBROUTINE state-	
23	ment.	
24	If execution of a procedure defined by a coroutine subprogram is completed by execution of a SUSPEND	
25	statement, and it is subsequently invoked by executing a RESUME statement while the same instance	
26	of its host exists as existed when the SUSPEND statement was executed, execution begins with the first	
27	executable construct following the SUSPEND statement that completed execution of the coroutine. Oth-	
28	erwise, when a procedure defined by a coroutine subprogram is invoked, whether by a CALL statement	
29	or a RESUME statement, execution begins with the first executable construct of the subprogram.	
30	[Editor: Replace “execution ... statement” by “completion of execution of an instance of a procedure”.]	394:0+12-13
31	[Editor: Replace “Execution ... statement” by “Completion of execution of an instance of a procedure”.]	415:6
32	[Editor: Replace “A ... statement” by “Execution of an instance of a procedure is completed”.]	415:8
33	[Editor: Replace “execution ... statement” by “completion of execution of an instance of a procedure”.]	423:24
34	8.2 One possible implementation strategy	
35	The main implementation problem is returning to the correct point of execution when a RESUME	
36	statement is executed. This can be accomplished by a hidden save variable to which the processor refers	
37	when control arrives in the procedure (recall that it is proposed to prohibit recursive coroutines). It is	
38	initialized by the processor to indicate the last exit was a result of a RETURN or END statement. If	
39	a SUSPEND statement is executed, the value of the hidden variable is changed to indicate that control	
40	should resume at the first executable construct after that SUSPEND statement. If control arrives in	
41	the procedure as a result of a CALL statement, or the hidden variable indicates that the last exit was a	
42	result of a RETURN or END statement, control proceeds to the first executable construct (recall that it	

1 is proposed that ENTRY statements not be allowed in coroutines). Otherwise, control proceeds to the
2 point indicated by the hidden variable. The hidden variable's use could be similar to an assigned GO
3 TO or to a computed GO TO, at the whim of the processor's developer.

4 A secondary problem is distinguishing whether control arrives in the procedure as a result of a CALL
5 statement or a RESUME statement. This can be accomplished in at least two ways. One is to have a
6 hidden argument that distinguishes the cases. The other is for the processor to generate two entry points,
7 with appropriately mangled names, with one referenced by CALL statements and the other referenced
8 by RESUME statements.

9 8.3 Inferior alternative

10 An inferior alternative is to allow an ENTRY statement within a construct other than WHERE, FORALL
11 or DO with *loop-control* consisting of *do-variable = scalar-int-expr, scalar-int-expr [, scalar-int-expr]*.
12 This is inferior because it puts the onus on the user to return to the correct place in the library code.
13 This could be ameliorated somewhat if the library routine has two layers — one that the user always
14 calls by the same name, which looks at a flag that controls what's going on and re-invokes the “real”
15 subroutine by the correct entry point. This slows things down. It is a step forward from the current
16 situation because it doesn't require to disrupt the control structure to implement reverse communication.
17 All in all, it's a relatively crappy solution.

18 9 History

19 This proposal was discussed and eventually rejected at meeting 166. The argument that led to its
20 rejection was that one could always put the extra information for user-defined code into an extensible
21 type. It was not considered at the time, however, that this requires the dummy argument of the
22 user-provided subprogram to be polymorphic, and that the user-provided subprogram must execute a
23 SELECT TYPE construct to gain access to the extra information. This overhead would not be necessary
24 in a coroutine interaction. Furthermore, type extension cannot be applied to iterator construction. It
25 was therefore brought up again at meeting 168, and rejected again, with the primary reason given being
26 “we want to think about it some more.” After gathering input from users, who urged resubmitting it,
27 it was resubmitted at meeting 169.

28 10 Examples

29 In the simplest case, to simulate coroutines with existing facilities, one would have an argument that
30 does dual duty to indicate what the caller is to do when the subroutine returns, and controls where
31 the subroutine goes with it is called. Since the subroutine may return from several places for the same
32 reason, the caller's decision-making process is messier than necessary. One usually instead has two
33 arguments, a **what** argument that tells the caller what to do, and a **where** argument that keeps track of
34 where the subroutine is to go when it gets control. The **where** argument is set by the caller to “start at
35 the beginning,” and if the caller changes it otherwise the guarantee is voided. The **where** argument can
36 be eliminated by using two entry points in the subroutine: one to start it up, which sets a local saved
37 **where** variable to “the beginning,” and the other to continue processing.

38 The next two pages give examples of simulating coroutines with current facilities, and doing it directly
39 using the proposed new facility. The simulation does not show whether **where** is an argument or a saved
40 local variable, as outlined above. Assume that appropriate stuff is saved, either explicitly or because
41 activation records are resurrected by RESUME statements.

```

1 10.1 Simulated
2   go to ( 20, 60, 90 ), where
3   ! Set up at beginning of problem, then
4   what = function ! "what" is a dummy argument
5 10  continue
6     if ( have enough function values for basic formula ) go to 30
7     ! Get ready for a function value for basic quadrature step, then
8     where = 1
9     return
10 20  continue
11     ! Add function * weight into quadrature estimate
12   go to 10
13 30  if ( error estimate is small enough ) then
14     where = 0
15     what = good enough
16     return
17   end if
18 40  if ( another formula doesn't exist ) go to 70
19 50  continue
20     ! if ( have enough function values for extended formula ) go to 30
21     ! Get ready for a function value for extended quadrature step, then
22     where = 2
23     return
24 60  continue
25     ! Add function * weight into quadrature estimate
26   go to 50
27 70  continue
28     ! Form a difference line of function values, then
29 80  continue
30     if ( nothing goofy in difference line ) then
31       ! Subdivide interval if it looks like smaller error will be
32       ! achieved (control structure for this not shown), or
33       what = done, but not as good as you asked for
34       where = 0
35       return
36     end if
37     if ( abscissa of difficult behavior sufficiently well isolated ) then
38       ! subdivide the interval -- control structure for this not shown
39     end if
40     ! Decide where to add a point to difference line to search for
41     ! difficult behavior, then
42     where = 3
43     what = function
44     return
45 90  continue
46     ! add function value to difference line
47   go to 80
48
49 ! Caller:
50   where = 0
51   do
52     call quadrature ( a, b, answer, tol, err, what, where )
53     if ( what /= function ) exit
54     ! evaluate function
55   end do

```

56 Aside from two IF ... THEN ... END IF blocks, this could be Fortran 66 code. The control flow is
57 "hiding" in the value of where.

```

1 10.2 With coroutines
2  ! Set up at beginning of problem, then
3  what = function ! "what" is a dummy argument
4  do while ( need a function value for basic formula )
5  ! Get ready for a function value for basic quadrature step, then
6  suspend
7  ! Add function * weight into quadrature estimate
8  end do
9  do
10 if ( error estimate is small enough ) then
11   what = good enough
12   return
13 end if
14 if ( another formula does not exist ) exit
15 do while ( need a function value for extended formula )
16 ! Get ready for a function value for extended quadrature step, then
17 suspend
18 ! Add function * weight into quadrature estimate
19 end do
20 end do
21 ! Form a difference line of function values, then
22 do
23 if ( nothing goofy in difference line ) then
24 ! Subdivide interval if it looks like smaller error will be
25 ! achieved (control structure for this not shown), or
26 what = done, but not as good as you asked for
27 return
28 end if
29 if ( abscissa of difficult behavior sufficiently well isolated ) then
30 ! subdivide the interval -- control structure for this not shown
31 end if
32 ! Decide where to add a point to difference line to search for
33 ! difficult behavior, then
34 what = function
35 suspend
36 ! add function value to difference line
37 end do
38
39 ! Caller:
40 call quadrature ( a, b, answer, tol, err, what )
41 do while ( what == function )
42 ! evaluate function
43 resume quadrature ( a, b, answer, tol, err, what )
44 end do

```

45 The control flow is obvious. Which would you prefer to write and maintain?